Lurch: a Lightweight Alternative to Model Checking

David Owen, Tim Menzies
Lane Department of Computer Science
West Virginia University
PO Box 6109 Morgantown, WV 26506-6109, USA
drobo75@hotmail.com, tim@menzies.us

Abstract

Formal methods, including model checking, is powerful but can be costly, in terms of memory, time, and modeling effort. Difficult problems, similar to the verification problem addressed by model checking, have been shown to exhibit a phase transition*, suggesting that an easy range of problem instances might be solved much faster and with much less memory using a new type of model checker based on partial, random search.*

Here we compare the performance of Lurch, our prototype random search model checker, to the popular tools SMV and SPIN. The tools' performance is compared for a range of randomly generated models based on a simple tic-tac-toe game. Our results suggest that Lurch might be used in place of existing tools for systems too large or too difficult to model small enough for conventional model checking.

1 Introduction

As software projects grow large and complex, as correctness grows more critical, and as more people are involved in the project, it becomes extremely important and extremely difficult to find and correct errors [7].

Complex systems can be modeled in the formal mathematical language of finite-state machines, and properties of these models can be automatically verified using *formal methods*—here we deal specifically with a subset of formal methods called *model checking* [8]. Model checking is a powerful tool, but it can be costly:

1. There is a *state-space explosion* problem: the input model is a system of interacting local finite-state machines; the single global finite-state machine constructed to represent all possible interactions of the local machines requires exponential (i.e., too much) memory, compared to the size of the input [4].

2. In order to minimize the global state space, the size and features of input models is restricted; in practice much time and effort goes into writing (and rewriting) input models, in part due to these restrictions [11].

During the last twenty years much progress has been made toward addressing these two issues. The two perhaps most significant state-space reduction techniques are *symbolic model checking* and *partial order reduction*. SMV, the *symbolic model verifier*, uses binary decision diagrams, or BDD's, to symbolically represent the global state space. This approach has been very successful in verifying models of synchronous (often hardware) systems [4]. SPIN [5], another very popular model checking tool, uses partial order reduction and specifically targets asynchronous (often software) models.

Still, the general problem remains, and various strategies—strategies that tend to increase the time, effort and expertise required to write input models—must be used to keep models as small as possible [1,4]:

1. **Compositional Reasoning**: the modular structure of some systems may be exploited; for example, we make certain assumptions about one part of the model in order to verify a property for another part; then we try to guarantee that the assumptions hold for the first part at any time the property must hold for the other.

2. **Abstraction**: we may be able to map the actual data values to a small set of abstract data values, for example.

3. **Symmetry**: we may be able to infer global properties from local properties in a model where many identical components interact with each other.

4. **Induction**: if we have an *invariant* process that represents the behavior of a family of processes, we can use induction to argue that any member process has some desired property already proven for the invariant.

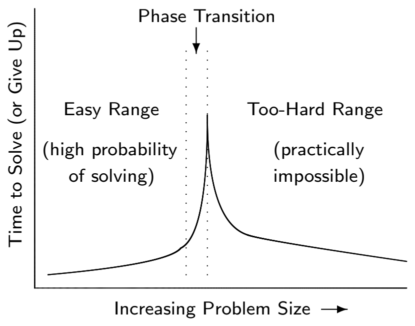


Figure 1. Hard problems (worst-case intractable) exhibit a phase transition.

Hard problems (worst-case intractable) like program verification exhibit a *phase transition* (figure 1): there is an *easy range* of problem instances and a range for which it is practically impossible to find a solution—but just a narrow band, the *phase transition* separaties these two regions [1]. We are working on an alternative to model checking, a tool that exploits the hypothetical *easy range* (figure 1), so that models can be verified quickly without using so much memory, and as a side benefit makes possible features that greatly simplify the modeling task.

How is it possible to exploit the *easy range*? We do this by using a partial, random search—partial, because unlike other model checking tools, the whole space of possible behaviors is not explored; random, because the choice of which behavior to explore is nondeterministic. For input models in the *easy range*, a random sample of possible behaviors will tell us nearly everything we could learn from an exhaustive search (for these models exhaustive search is simply overkill). And for models in the *too hard* range, where the partial, random search is not effective, exhaustive search is intractable.

Or, as stated by Knuth in more general terms, "No matter what complicated thing you have, . . . there's a fairly good chance that random sampling will give insight. Of course if you start with purely random data [e.g., the worst-case end of the the *too hard* range], then random sampling is going to tell you that it is purely random data" [3].

Elsewhere similar ideas about how program structure determines whether a verification problem falls into the *easy range* are expressed in terms of *funnels* or *small backbones* [12]. A *funnel* is a small set of key variables that largely determine the behavior of the entire system; if an execution path stumbles into the *funnel*, it will nearly always be directed to the same behavior. *Easy range* models are easier to verify (and arguably more testable [1, 2]) because they contain *funnels*. In these models a comparatively small amount of exploration is likely to quickly find the key variables and therefore everything its possible to find.

In this paper we describe a simple random search model checking tool, a tool which, if the *phase transition* and *funnel* ideas apply, should work nearly as well as more sophisticated tools like SMV and SPIN.

Section 2 describes *Lurch*, which is the current C implementation of our random search model checking tool. Section 2.1 explains Lurch is able to represent large systems without exponential memory, and section 2.2 how we are able to find solutions without exponential time. Section 3 describes the experiment presented in this paper. In section 3.1 we describe how to model a simple problem based on tic-tac-toe games using finite-state machines (the formal basis for model checking input languages). In section 3.2 we compare Lurch's performance, on tic-tac-toe problems, to two widely used model checkers, SPIN and SMV. The concluding section discusses our results and describes our continuing work on Lurch.

2 Lurch

2.1 Efficient AND-OR Graph Representation

The *state-space explosion* encountered by model checking tools is due to the exponential size of the global finite-state machine used to represent all possible interactions of the local machines in the input model [4]. Figure 2 shows how it is possible to construct a more efficient AND-OR graph representation of the global state space. The translation procedure assumes that finite-state machines have transitions defined as follows: each transition begins in the *current state*, and takes place if all *inputs* are true. If a transition takes place, all outputs are set true and the machine moves to the *next state*. Figure 3 shows a very simple example. For more on the AND-OR graph representation, see [1, 10, 11].

In an AND-OR graph, an AND-node is considered true if all of its parent nodes are true; an OR-node is considered true if any one of its parents is true. We

```
1: for (each local machine) do
2:    for (each state) do
3:       Make an OR-node; connect it with a NO-edge
           to each OR-node representing another of this
           machine's states.
4:    for (each transition in this local machine) do
5:       Make an AND-node; connect it with a NO-edge
           to each AND-node representing another of this
           machine's transitions.
6:       Make current state a YES-edge parent of the
           AND-node;
7:       Make input(s) (a) YES-edge parent(s) of the
           AND-node;
8:       Make next state a YES-edge child of the
           AND-node;
9:       Make output(s) (a) YES-edge child(ren) of the
           AND-node.
```

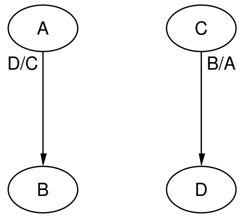
Figure 2. Constructing an AND-OR graph to represent the global state space.

add NO-edges to indicate which nodes may not be true at the same time. For example, since a local machine may not be in two states at once, NO-edges connect all OR-nodes representing states in the same local machine (figure 2, line 3). Also, since two transitions in a local machine may not occur simultaneously, NO-edges connect all AND-nodes representing transitions in the same local machine (figure 2, line 6). The resulting AND-OR graph has $O(n)$ nodes and $O(n^2)$ edges, where n is the size of the input [1].

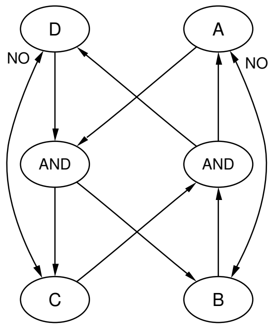
2.2 A Fast Random Search Algorithm

Unfortunately, complete search of our AND-OR graphs requires exponential time [1]. Figure 4 shows the fast partial, random search procedure Lurch uses to search AND-OR graphs. As stated briefly in the introduction, the search procedure is partial in that there is no guarantee that the entire AND-OR graph will be explored. Each iteration (of the *search* function, beginning on line 20 in figure 4) finds one global state path. With many iterations it becomes likely that nearly all of the reachable state space is explored. The procedure is random in that, when two or more paths may be explored, the choice is made based on the order nodes are popped from the queue (line 7). Since nodes are always pushed at a random index (lines 19, 24, 29, 34), the choice of which path to explore is nondeterministic.

For each node (Figure 4, line 1), *kids* and *NO-kids* (line 2) are lists of children via normal and NO-edges. The *disqualified*, *frontier*, and *reached* fields (line 3) mark at what time during the search a node is disqualified, part of the frontier, or reached; *wait* (line 4) is an integer indicating how may parents still need to be reached before the node is reached—*wait* is initialized to 1 for an OR-node and, for an AND-node, to the number of parents it has.



Finite-state machines: each has two states and a single transtion, labeled *input*/*output*.



AND-OR graph: OR-nodes are labeled according to the local state each represents; there is an AND-node for each local transition above. NO-edges join mutually exclusive nodes.

Figure 3. An AND-OR graph representing the interaction of two finite-state machines.

Output from the search procedure is a series of global state paths. For each path, there is a time value associated with each global state in the path. Each global state has a value for all local states in the input model. The search is used to tell us which local states were reached, and what time each was reached. So it is possible to include temporal logic relationships as part of the model, and to prove that local states representing the satisfaction or violation of temporal logic properties are reachable, even without exploring the entire global state space. In this way our technique should be able to provide nearly the same functionality as a model checking tool—provided that the partial search is nearly complete. The experimental results presented in section 3.2 show that Lurch performs very well in practice.

```
node {
kids, NO-kids;
disqualified, frontier, reached;
wait; }
process-queue(time) {
while ($Q \neq \emptyset$) do
   n $\leftarrow$ pop(Q);
   if (n.disqualified $\neq$ time) then
      if (n is an OR-node) then
         n.reached = time;
         for ($\forall$ nodes $n' \in$ n.NO-kids) do
            $n'$.disqualified = time;
      for ($\forall$ nodes $n' \in$ n.kids) do
         $n'$.wait $\leftarrow$ $n'$.wait $-$ 1;
         if ($n'$.wait = 0) then
            if ($n'$ is an OR-node) then
               $n'$.frontier = time;
            else if ($n'$ is an AND-node) then
               Q $\leftarrow$ $n'$ at random index; }
search() {
time = 0;
while (NOT(path-end or cycle)) do
   for ($\forall$ nodes n : n.frontier = time $-$ 1) do
      Q $\leftarrow$ n at random index;
      for ($\forall$ nodes $n' \in$ n.NO-kids) do
         $n'$.disqualified = time;
   for ($\forall$ nodes n : n.reached = time $-$ 1) do
      if (n.disqualified $\neq$ time) then
         Q $\leftarrow$ n at random index;
   process-queue(time);
   time $\leftarrow$ time + 1; }
main() {
for (i = 1 to MAX-PATHS) do
   for ($\forall$ nodes n) do
      n.disqualified $\leftarrow$ n.frontier $\leftarrow$ n.reached
        $\leftarrow$ UNDEF;
      reset n.wait to initial value;
   for ($\forall$ nodes n : n is true initially) do
      Q $\leftarrow$ n at random index;
   search(); }
```

Figure 4. Partial, random search procedure for AND-OR graphs.

2.3 When to Stop (saturation)

Lurch is implemented as a Monte Carlo algorithm; that is, the basic procedure is repeated and, the longer it runs, the more accurate the result will be. But when are the results accurate enough? When is it okay to stop?

Figure 5 shows typical output from Lurch running on a large finite-state machine input model. As Lurch runs, it explores the reachable state space, at first finding nearly all new information, but after a little while most of Lurch's findings are redundant; figure 5 illustrates this: the percentage of information which is new (vs. redundant) starts out at 100 %, but very quickly decreases to near zero. We use this quick *saturation* effect in Lurch output (see [11]) to tell when to stop: when some set *saturation point* (close to 0 %) is reached, we assume that Lurch is unlikely to find any more interesting information.

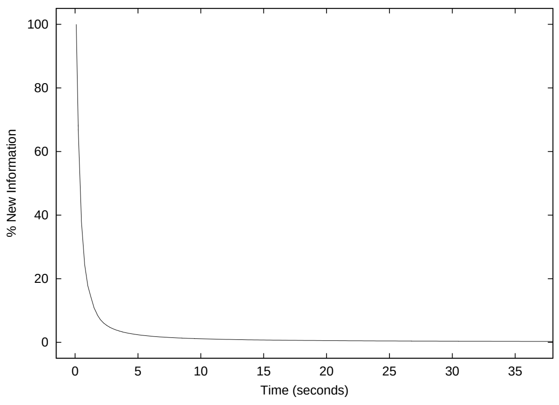


Figure 5. Typical Lurch output for a large model—quick saturation.

In order to determine, while running, what percentage of its results are new, Lurch stores a hash value for each global state it finds. When Lurch attempts to store a value, if the value is not already stored, the global state associated with it is considered new information; otherwise it is considered redundant. In order to save memory the user can limit the number of global states stored. Because of this hashing scheme, some of what Lurch considers redundant is actually new information. So the user may manipulate how much memory Lurch uses and how accurate the search results are by adjusting parameters at the command line. For the experiments presented in this paper, Lurch was run at or very close to its default settings, resulting in maximum run times of about one minute, close to the same amount of time required by SMV and SPIN.

2.4 A Convenient Input Language

Lurch's time and memory saving innovations make it possible to add a convenient extenstion to our finite-state machine input models: transitions may call functions written in ordinary C code (a new feature added since earlier work reported in [1, 11]). We expect this will significantly reduce the modeling effort required for many verification tasks. Also, the fact that functions called by finite-state machines are written in C (and not some other language) is simply because lurch is implemented in C; that is, an Ada, C++, or Java implementation of lurch would allow Ada, C++, or Java functions.

This addition requires just slight changes to the search procedure in figure 4. Before processing a node's children (line 13), if that node is an AND-node, the C function associated with the input for the transition represented by the AND-node is called. Only if it returns *TRUE* are the AND-node's children processed. Also, in addition to processing the children, the C function associated with the output of the transition represented by the AND-node is called.

3 Tic-Tac-Toe

3.1 The Problem

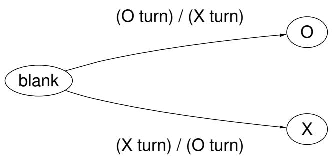


Figure 6. Finite-state machine representing one space on a tic-tac-toe board (edges, which represent transitions, are labeled input / output).

Here we use a range of finite-state machine models to compare the performance of Lurch to the popular model checking tools SMV [6] and SPIN [5]. The problem is set up as follows: for an $n \times n$ board size tic-tac-toe game already in progress, is it still possible for either player to win (complete a row of n X's or O's)?

Why tic-tac-toe? Because it's easy for a person to look at the board for a game in progress and determine whether it's still possible for one of the players to win. You would just check that all of the horizontal, vertical, and two diagonal rows contain both X's and O's; if so, obviously neither player can win. But for lurch (or SMV or SPIN) there is no easy solution oracle. The tools must actually simulate possible sequences of play until they find a winner.

Figure 7 shows a very simple 2×2 tic-tac-toe board for a game already in progress. Figure 8 shows the Lurch input model for the board shown in figure 7.

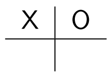


Figure 7. A 2 $\times$ 2 tic-tac-toe game in progress (obviously it's still possible for someone to win—in fact it must always be possible for 2 $\times$ 2 boards).

The Lurch input model (figure 8) is divided into two parts. As mentioned in section 2.4, the Lurch input begins with ordinary C code, declaring global variables and defining functions. After the double percent (%%), finite-state machines are defined, each separated by one or more blank lines. There is one four-column line for each transition (columns are separated by semicolons). The first column is the current state, the second a list of inputs, the third a list of outputs (side effects of the transition), and the fourth the state that the machine goes to after the transition occurs. The first state listed in each machine is considered true initially. A particular state can be explicitly declared true initially by putting it on a line alone at the beginning of a new finite-state machine definition.

In figure 8, the first four finite-state machines represent the four spaces on the tic-tac-toe board from figure 7. Spaces (1,2) and (2,2) are initially X and O, so the second and fourth finite-state machines begin by explicitly listing these as initial states. The fifth finite-state machine represents the property: *it is still possible for one of the players to win.* Its initial state is *no-win*, but if any combination of other machines' states indicates that a player has won the game, this machine transitions into the state *win*.

3.2 Comparison of Lurch, SMV, and SPIN

To compare the performance of Lurch to SMV and SPIN, we generated 20 random boards of each size, from 2×2 up to 15×15 spaces. Figure 9 shows the SMV input model for the tic-tac-toe problem show in figure 7; figure 10 shows the same problem written in Promela, the input language for SPIN. Models are written a little bit differently for each of the three tools, but are of approximately the same length and complexity. For more on the SMV input language, see [6]; for more on Promela, the input language for SPIN, see [9].

Figures 11, 12, and 13 compare Lurch to SMV and SPIN (in two different modes), showing the time and memory required, and the accuracy achieved by, each tool. Each value plotted represents the average value

```
enum { UNDEF, X, O } turn = X;

int x_turn() {
  if (t != X) return FALSE;
  t = UNDEF; return TRUE;
}

int o_turn() {
  if (t != O) return FALSE;
  t = UNDEF; return TRUE;
}

%%

s1_1-blank; (x_turn()); {t = O;}; s1_1-X;
s1_1-blank; (o_turn()); {t = X;}; s1_1-O;

s1_2-X;
s1_2-blank; (x_turn()); {t = O;}; s1_2-X;
s1_2-blank; (o_turn()); {t = X;}; s1_2-O;

s2_1-blank; (x_turn()); {t = O;}; s2_1-X;
s2_1-blank; (o_turn()); {t = X;}; s2_1-O;

s2_2-O;
s2_2-blank; (x_turn()); {t = O;}; s2_2-X;
s2_2-blank; (o_turn()); {t = X;}; s2_2-O;

no-win; s1_1-X,s1_2-X; -; win;
no-win; s2_1-X,s2_2-X; -; win;
no-win; s1_1-X,s2_1-X; -; win;
no-win; s1_2-X,s2_2-X; -; win;
no-win; s1_1-X,s2_2-X; -; win;
no-win; s1_2-X,s2_1-X; -; win;
no-win; s1_1-O,s1_2-O; -; win;
no-win; s2_1-O,s2_2-O; -; win;
no-win; s1_1-O,s2_1-O; -; win;
no-win; s1_2-O,s2_2-O; -; win;
no-win; s1_1-O,s2_2-O; -; win;
no-win; s1_2-O,s2_1-O; -; win;
```

Figure 8. Figure 7 as a Lurch input model.

```
MODULE space(turn, init_val)
VAR
  val : {blank, X, O};
ASSIGN
  next(turn) := case
    (val = blank & turn = X) : O;
    (val = blank & turn = O) : X;
    1 : turn;
    esac;
  init(val) := init_val;
  next(val) := case
    (val = blank & turn = X) : X;
    (val = blank & turn = O) : O;
    1 : val;
    esac;

MODULE main
VAR
  s1_1 : process space(turn, blank);
  s1_2 : process space(turn, X);
  s2_1 : process space(turn, blank);
  s2_2 : process space(turn, O);
  turn : {X, O};
ASSIGN
  init(turn) := X;
SPEC
  AG(!((s1_1.val = X & s1_2.val = X) |
       (s2_1.val = X & s2_2.val = X) |
       (s1_1.val = X & s2_1.val = X) |
       (s1_2.val = X & s2_2.val = X) |
       (s1_1.val = X & s2_2.val = X) |
       (s1_2.val = X & s2_1.val = X) |
       (s1_1.val = O & s1_2.val = O) |
       (s2_1.val = O & s2_2.val = O) |
       (s1_1.val = O & s2_1.val = O) |
       (s1_2.val = O & s2_2.val = O) |
       (s1_1.val = O & s2_2.val = O) |
       (s1_2.val = O & s1_2.val = O)))
```

Figure 9. Figure 7 as an SMV input model.

for 20 randomly-generated tic-tac-toe games already in progress. For these plots Lurch was run at or near the default parameters for stopping criteria (see section 2.3). SMV was run in its normal complete search mode, but restricted to only the reachable state space (the *-f* option). SPIN was run first in normal complete search mode and then in its *supertrace* approximation mode, which uses much less memory and, at least for our experiments, produces at least as accurate results. Tools were used on input models of increasing board size until they ran out of memory (SMV plots only go to 9 × 9; SPIN in normal mode only to 12 × 12). The experiments were done on a DELL Inspiron with a 1.8 GHz processor and 512 MB of RAM, running Windows XP Professional.

In figure 11, time for SMV spikes over one minute for 9 × 9 boards, SPIN (in normal complete search mode) reaches about 20 seconds for 8 × 8 boards and remains there up to 12 × 12. SPIN in supertrace approximation mode requires about the same amount of time as SPIN in normal mode, up to 9 × 9 boards, and then continues to increase, reaching nearly 90 seconds for 15 × 15 boards. Lurch, using default (or near default) settings for stopping criteria, remains very fast for boards up to 10 × 10, and then reaches about one minute for 15 × 15 boards. Although it may be difficult to see on the graph, Lurch is fastest for all runs in which any tool took more than 5 seconds.

Figure 12 compares the amount of memory used by Lurch, SMV and SPIN. To get fair memory information, Windows XP resource use utilities were used to log *Working Set Peak* for each tool. Logs were updated only once per second, so memory use is not plotted for board sizes so low that the tool ran too quickly to get good memory data.

In figure 12, SPIN (running in normal mode) reaches past 250 megabytes for 8 × 8 boards, and continues at around 260 megabytes through board size 12 × 12. One convenient feature of SPIN is that it runs out of memory *gracefully*—there is an error message and the program terminates. So in figure 12, from board size 8 × 8 to 12 × 12, the SPIN plot is actually showing: *how much memory was used before SPIN gave up*; likewise, in figure 11, for board size 8 × 8 to 12 × 12, the SPIN plot shows how much time it took SPIN to run out of memory. SMV, at least in the close-to-default

```
mtype = {blank, X, O};
mtype s1_1 = blank, s1_2 = X;
mtype s2_1 = blank, s2_2 = O;

init {
  mtype turn = X;
  end : do
    :: (s1_1 == blank && turn == X) ->
         atomic { turn = O; s1_1 = X }
    :: (s1_1 == blank && turn == O) ->
         atomic { turn = X; s1_1 = O }
    :: (s1_2 == blank && turn == X) ->
         atomic { turn = O; s1_2 = X }
    :: (s1_2 == blank && turn == O) ->
         atomic { turn = X; s1_2 = O }
    :: (s2_1 == blank && turn == X) ->
         atomic { turn = O; s2_1 = X }
    :: (s2_1 == blank && turn == O) ->
         atomic { turn = X; s2_1 = O }
    :: (s2_2 == blank && turn == X) ->
         atomic { turn = O; s2_2 = X }
    :: (s2_2 == blank && turn == O) ->
         atomic { turn = X; s2_2 = O }
    od;
}

never {
  T0_init: if
    :: (s1_1 == X && s1_2 == X) ||
       (s2_1 == X && s2_2 == X) ||
       (s1_1 == X && s2_1 == X) ||
       (s1_2 == X && s2_2 == X) ||
       (s1_1 == X && s2_2 == X) ||
       (s1_2 == X && s2_1 == X) ||
       (s1_1 == O && s1_2 == O) ||
       (s2_1 == O && s2_2 == O) ||
       (s1_1 == O && s2_1 == O) ||
       (s1_2 == O && s2_2 == O) ||
       (s1_1 == O && s2_2 == O) ||
       (s1_2 == O && s2_1 == O) ->
         goto accept_all
    :: (1) -> goto T0_init
    fi;
  accept_all: skip
}
```

Figure 10. Figure 7 as a Promela model (input for SPIN).

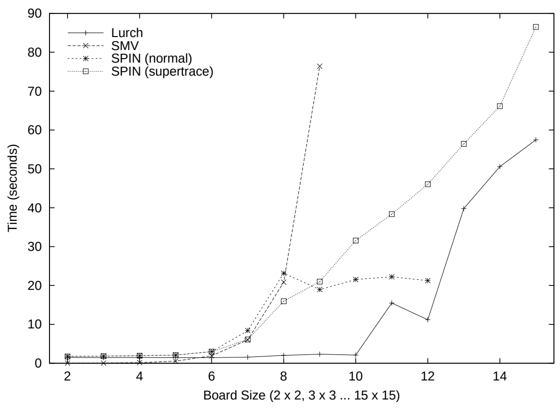


Figure 11. Time comparison for Lurch, SMV, and SPIN (2 modes) running on tic-tac-toe boards of increasing size.

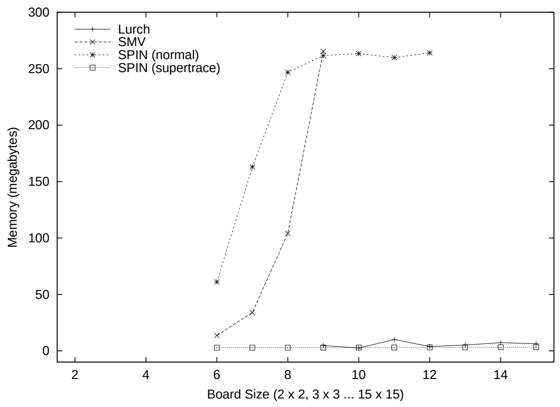


Figure 12. Memory comparison for tic-tac-toe experiment.

mode we used for this experiment, did not run out of memory so gracefully. Figure 12 shows a spike around 260 megabytes for 9×9 boards. For any larger boards, SMV required a lot of virtual memory and therefore ran so slowly it was not practical to continue. Figure 12 shows (bottom right) that Lurch uses about the same amount of memory as SPIN running in its supertrace approximation mode, between 10 and 20 megabytes for boards up to 15×15.

Figure 13 compares the accuracy (in terms of % error) of Lurch, SMV and SPIN in this experiment. Although it is difficult to see, SMV is accurate (0 % error) from board size 2×2 through 9×9, at which point it was no longer used because it ran out of memory for larger boards. When SPIN (in normal mode) reaches 7×7 and begins to run out of memory for some of the runs, its % error quickly rises, so that for boards from 8×8 through 12×12, it is not very reliable. SPIN in supertrace mode behaves similarly, but is somewhat reliable up to boards of size 10×10.

Only Lurch continues to be reliable through 15×15 boards, at which point we get the first *false positive*—that is, Lurch's partial search reached the predefined saturation point without detecting the error, that it was indeed possible for a player to win the game.

4 Conclusion

In comparison to SMV and SPIN, Lurch performed very well for these experiments. Currently the only drawback to using Lurch is the possibility, however unlikely, of a false positive result. When SMV must be terminated early because it runs out of memory, it is

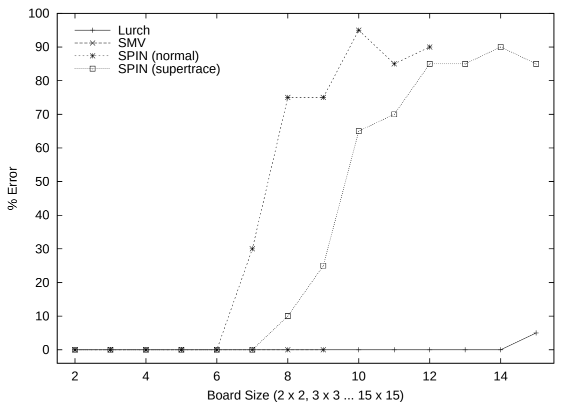


Figure 13. Accuracy comparison for tic-tac-toe experiment.

obvious to the user that the verification has failed. And SPIN, in both normal and supertrace modes, when it runs low on memory, warns the user that the result cannot be trusted. But Lurch, because it is based on a partial search, can not provide the same level of certainty. Because of this we do not suggest Lurch's strategy as a replacement for complete model checking methods, where they succeed, i.e., boards up to around 8 × 8. But for problems where model checking would require too much memory or too much modeling effort, i.e., boards over 9 × 9, we believe Lurch will prove to be a valuable addition to current model-based verification tools.

We are now working on a new experiment to compare Lurch and SMV's performance on formal models of a large flight guidance system. We hope to see the same level of accuracy, and the same minimal time and memory requirements seen in these tic-tac-toe experiments.

References

[1] David Owen. Random Search of AND-OR Graphs Representing Finite–State Models. Master's thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia University, 2002.

[2] David Owen, Tim Menzies, and Bojan Cukic. What Makes Finite-State Machines More (or less) Testable? In *Proceedings of IEEE International Conference on Automated Software Engineering (ASE)*, 2002.

[3] Donald Knuth. *Things a Computer Scientist Rarely Talks About.* Stanford Center for the Study of Language and Information, 2001.

[4] Edmund A. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, Cambridge, MA, 1999.

[5] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[6] K. L. McMillan. The SMV System, 2000. Available at `http://www-cad.eecs.berkeley.edu/~kenmcmil/`.

[7] Michael A. Friedman and Jeffrey M. Voas. *Software Assessment: Reliability, Safety, Testability.* John Wiley & Sons, 1995.

[8] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems.* Cambridge University Press, 2000.

[9] Rob Gerth. Concise Promela Reference, 1997. Available at `http://spinroot.com/spinMan/Quick.html`.

[10] Tim Menzies, Bojan Cukic, Harhsinder Singh, and John Powell. Testing Nondeterminate Systems. In *ISSRE 2000*, San Jose, CA, 2000.

[11] Tim Menzies, David Owen, and Bojan Cukic. Saturation Effects in Formal Verification. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2002.

[12] Tim Menzies and Harhsinder Singh. Many Maybes Mean (Mostly) the Same Thing. In *2nd International Workshop on Soft Computing Applied to Software Engineering*, Netherlands, 2001.